1. Overview
The purpose of this paper is to give a progress report on the ongoing work on the project. This paper is organized as follows. Section 2 details the changes that have been made from the project specifications described in the project proposal. Section 3 describes the structure of the compiler. Section 4 describes the implementation details concerning the SUIF2 compiler. Section 5 presents the work done at this phase of the project. Section 6 details the schedule for the rest of the project.

2. Project Changes
It became clear during the low level design phase of the project that the original scope of the project was too large to finish within the allotted time. We found that the only way to be able to simplify the project was to restrict the use of the object fusion optimization to only certain data structures defined in a strict manner, which will be described in Section 3 below.

Originally, our goal was to transform a general tree-like data structure into the hyperobject representation of the tree. However, we found that we need to restrict the structure layout of the tree node. Similarly, in order to implement the data traversal algorithm for this new hyperobject data structure, we need to assume the additional restriction that the user implement the traversal function with two procedures: the get_key() and the next_node() functions. Once again, these will be described in much more detail in the next section. Moreover, the user is allowed to access the Key and Next fields only during traversal, and the traversal function returns a pointer to the data field.

Note that despite the restrictions imposed, our work can nonetheless serve as a proof-of-concept model. The extensions to regain generality are straightforward, and consist of defining a meta-language through which the user passes information to the compiler about the functions used to traverse the data structure and the fields of the structure’s elementary nodes. The issue of receiving information from the user is orthogonal to the use of that information to perform object fusion. Due to lack of time we decided to focus on the latter, and defer work on the former.

3. Object Fusion Compiler Details
3.1 Structure Transformation
The first thing that needed to be done is the compaction of multiple tree node structures into a single hyperobject node structure. Figure 1 shows an example of the tree data structure before object fusion is performed on the code. In the figure, all the encircled nodes can be copied into a single hyperobject node.

We restrict the programmer to a very strict node structure, which is shown in Figure 2. The tree node structure can only contain three fields: the Key, an array of the Next nodes, and the data that this node contains. The names of these fields are not restricted, although the order that these fields are declared is. The reason for this is that different programmers have different methods of implementing tree nodes. We need to restrict the layout of the tree node structure in order to be able to distinguish between the different fields.

![Figure 1: Tree structure. Object Fusion optimization creates a single hyperobject node containing all the data of the encircled nodes.](image1)

```c
struct treenode {
    <type1> Key;
    struct treenode *Next[c];
    <type2> Data;
};
```

![Figure 2: Original tree node structure layout.](image2)

The problem with the node layout of Figure 2 is that by placing the Key, Next, and Data fields in consecutive memory addresses, the keys of the children nodes may reside in a different cache line.
Thus, in order to traverse through the tree of Figure 1, multiple cache accesses may need to be performed.

Figure 3 shows the code representation of the new hyperobject data structure. N is the number of tree nodes from the original data structure that are contained in the hyperobject. However, we only want to put full levels of the tree into a hyperobject, so the calculation of N is actually a little more complicated than that. It can be described with the equation below:

\[
N = \max \left\{ \sum_{i=0}^{H} c^k \right\}, \text{ s.t. } N \leq \frac{\text{size(cache line)} - \text{size(bitmask)}}{\text{size(key)}}
\]

The equation above assigns to N the maximum number of nodes whose keys can fit in a single cache line along with the valid bitmask, and comprise entire levels of the subtree. For instance, in the tree of Figure 1, the number of nodes that would fit into a hyperobject would either be 4, 13, 40, and so on. The reason for this is that we do not want to deal with having a partial subtree level in one hyperobject and the rest of the level in another hyperobject. That would be too complex to keep track of in the framework of a class project.

H is the height of the subtree contained within a hyperobject. The number of pointers exiting the hyperobject is \(c^H\) where c is the tree’s branching factor. These represent the pointers seen exiting the dotted area in Figure 1. The valid field is a bitmask of N bits which tells us which key entries in the hyperobject are valid.

The hyperobject node code representation of Figure 3 results in the data layout shown in Figure 4. \(K_i\) are the keys of the nodes contained in the hyperobject. \(V_i\) is the valid bit for node i. It is 1 if \(K_i\) is valid and 0 if \(K_i\) is invalid. \(P_j^i\) represents child j of node i. These are the pointers exiting the hyperobject, so only the pointers of the “leaves” of the hyperobject are of any significance. \(D_i\) is the data associated with node i.

Since in the hyperobject the tree node keys are now all contained within a single cache line, they can all be retrieved with a single memory access, thus reducing the amount of time needed to traverse through the tree structure.

3.2 Traversal Function Transformation

The transformation of the traversal algorithm is the most complicated part of the project so far. The problem is that since programmers have many different coding styles, it is very difficult for the compiler to parse through a code snippet and automatically extract all the required information.

Thus, once again, we impose restrictions to the coding style of the programmer. A code snippet for the original traversal function is given in Figure 5 below.

```c
struct new_node {
    <type1> Key[N];
    bit valid[N];
    struct new_node *next[c^H];
    <type2> Data[N];
};

p = top_ptr; // ptr to top of data struct
while (p != NULL) {
    unsigned int j;
    key = get_key(p);
    if (key == val)
        return p->data;
    else if (key < val)
        j = 1;
    else if (key > val && key < val*2)
        j = 2;
    else
        j = 3;
    p = next_node(p, j);
} return NULL;
```

Figure 5: Tree traversal function prior to running Object Fusion optimization on it.
The programmer needs to implement two functions: `get_key()` and `next_node()`. The `get_key()` function extracts the key from the original tree node data structure. The body of the `get_key()` function would be something like:

```c
return p->key;
```

After object fusion, the code would be transformed into an array access:

```c
return p->key[i];
```

The `next_node()` function in the original C source code returns a pointer to the next tree node to traverse. This function would be modified to return an index to the next node to retrieve from the hyperobject node instead. Let \( c \) be the branching factor of the tree structure. In the transformed code, if we are at node \( i \) within the hyperobject and we want to access its \( j \)-th child, then:

\[
next\_node(i, j) = \sum_{k=0}^{L_i} c^k - i - 1 + \sum_{k=0}^{L_i-1} c^k \cdot c + (j-1) + 1
\]

where \( L_i \) is the subtree level of node \( i \). The expression \( E_1 \) represents the number of nodes that need to be skipped at level \( L_i \). The expression \( E_2 \) represents the number of tree nodes that need to be skipped at the next level of the subtree \( (L_i+1) \). Finally, the expression \( E_3 \) determines the number of siblings of the next node that need to be bypassed.

In addition to the two functions mentioned, function `next_ptr()` is created by the compiler. It returns an index to the next pointer array of the hyperobject. It is called when we visit another hyperobject. In that case `next_node()` returns a value larger than \( N \), and `next_ptr()` returns:

```c
next\_ptr(i, j) = E_2 + E_3 = \sum_{k=0}^{L_i-1} c^k \cdot c + (j-1)
```

The modified traversal algorithm is shown in Figure 6.

### 4. SUIF Implementation Details

Since object fusion is a high-level optimization, we chose to implement it in SUIF2, which gives us the ability to manipulate high-level constructs like loops and structure fields.

We use selective walkers to traverse the compiler's high level IR and perform the transformations. First we calculate the number of objects that will be fused into a hyperobject. Then we manipulate the fields of the original data structure using methods of the `SuifObjectFactory`, `BasicObjectFactory`, and `TypeBuilderFactory` classes. In the process, we fix the field's bit offset, size, and alignment attributes, and replace the original data type with an array type. This ensures that all fields are placed in memory space similarly to Figure 4. The valid bitmask is also created, attached to the group's symbol table, and inserted into the appropriate position. The structures are annotated after the manipulation, to prevent revisiting them in the future. In order to detect bugs early in the process, at the end of the object fusion pass all types are checked, the structure of the IR representation is validated, and all invariants are enforced (in separate passes).

```c
p = top_ptr; // ptr to top of data struct
while (p != NULL) {
    unsigned int j;
    for (i = 0; p->valid[i] && i < N) {
        key = get_key(i);
        if (key == val)
            return p->data;
        else if (key < val)
            j = 1;
        else if (key > val && key < val*2)
            j = 2;
        else
            j = 3;
        i = next_node(i, j);
    }
    p = p->next[next_ptr(i, j)];
}
return NULL;
```

**Figure 6: Hyperobject traversal function after running Object Fusion optimization on the code.**

The code transformations are done in a similar manner, utilizing the factory classes and the utilities headers. Accesses to the original structure fields are transformed into array accesses, and the body of the original traversal loop is attached to the newly inserted for-loop that traverses the objects within the hyperobject. The `next_node()` function call is also modified appropriately, since it now contributes to the traversal of the hyperobject. To facilitate moving across hyperobjects, a new statement is inserted at the end of the for-loop which calls the compiler generated function `next_ptr()`.

### 5. Accomplishments

We have sketched exactly how the optimization will be implemented, including insertion and deletion, along with the necessary methods to support our needs. We have also defined the information the user should pass to the compiler. We have implemented the hyperobject data structure transformations which work. We are currently working on the traversal code transformations. The optimization is implemented as a SUIF2 pipelinable pass, and creates a DLL with the fusion module registered, which can be invoked via the suifdriver interface.

#### 5.1 Plan of Work

1. Finish implementation of data structure traversal algorithm. Prefetch cache lines holding pointers to other hyperobjects when a hyperobject is accessed.
2. Implement insertion and deletion and bug fixing.